Documentation for the CS441 Parser Assignment

## LLM Choice:

As with my first project, I have chosen Claude as my LLM. I was highly impressed by its performance on the first project and will seek to keep using it.

## Author Note:

I am assuming that you have access to the Claude chat logs and will reference them throughout this. Any point where I reference Claude can be verified through the submitted chat logs. I did not attach screenshots as this document is rather long already

## Development Process:

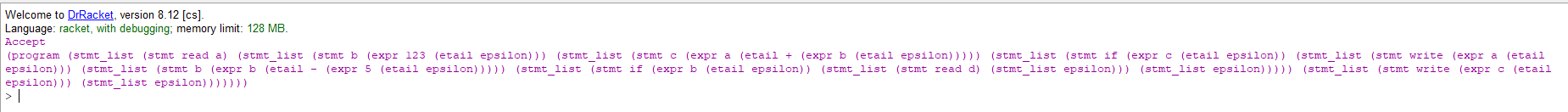
First, I prompted Claude to provide code for the designated grammar, using a function to represent each of the nonterminals. There were some minor issues (such as adding parentheses) that were able to be fixed manually. It also didn’t add (parse “input.txt”) which was weird, but I added it and it worked!

After I ran the original code, I noticed that it was unable to detect parentheses in the testing files, so I prompted Claude. It denoted new tokens, “lparen”, “rparen”, and “assign”. (, ), and = respectively.

After running the new code, I noticed that it errored out any test file with not having “eof” at the end. So I asked it to correct this and it gave me the test file test100.txt.

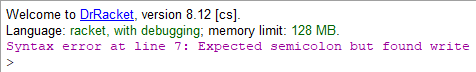
At this point, everything went great! I finally got back my first parse tree seen below. This worked perfectly.

**Test with “test100”**



Now, it was time to check on the sample files given to us through the canvas announcement (file1, file2, file3, and file4).

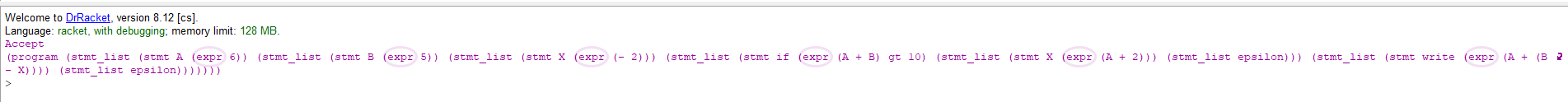
**Test with “file1”**



The announcement said file1 should run properly, but it didn't. I double checked the grammar and my code was definitely correct, semicolon should appear after endif. I waited a couple days and received the errata announcement clarifying my code was, in fact, correct!

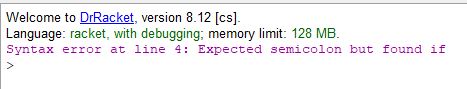
With that good news, I updated file 1 and finally got my parse tree!

**Test with Updated “file1”**



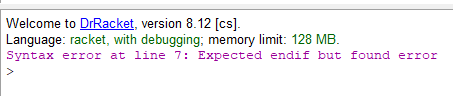
I dislike the list formatting for parse trees (find it difficult to read), but it is very easy on the developer to print out. Also, the list format was specifically requested per assignment instructions so that looks good! I’m very confident that this is a recursive descent parse tree, but I feel as though there’s some stuff missing. It does cover the full code, but it doesn’t turn A or B into id’s nor 6 or 5 into num/numsign. Unsure of what it's missing, but for now, let's check the rest of the sample files!

**Test with “file2”**



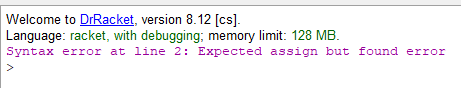
Now this is kind of “half correct” or “technically correct”? The code is doing the right thing. The missing semicolon is found on line 3, but the found if is on line 4. Still, I am very happy with this outcome. Correct syntax error, correct location.

**Test with “file3**



Perfect!

**Test with “file4”**



Perfect x2!

Also in the errata there was a mention of adding an additional line to the grammar; The new nonterminal “compare”.

compare -> < | <= | > | >= | == | !=

I implemented this by prompting Claude which had some, again, minor errors that were fixed manually (Parentheses and some formatting). I ran the test files after implementing this and it seemed to work great!

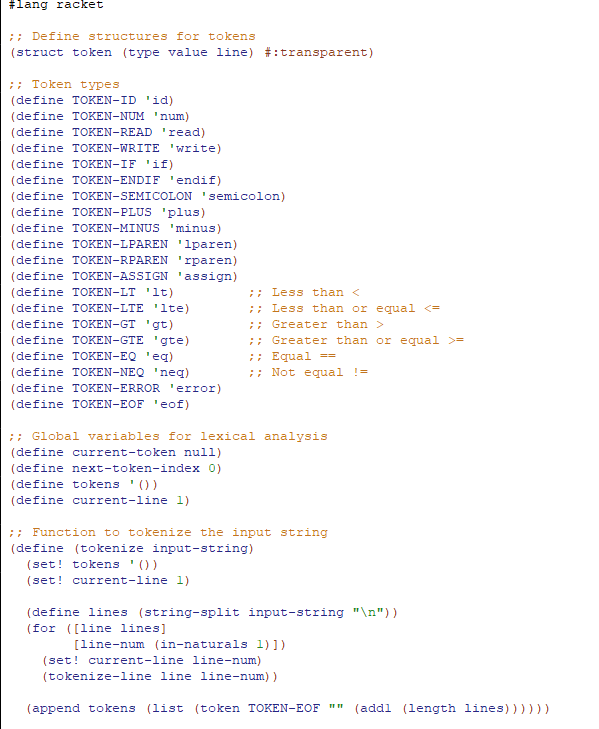
I got communication from the professor that the parse trees it generated for test100 and file1 were correct! With this in mind my last goal was to generate automated testing.

I went into a new chat log with Claude due to running out of space on my previous one. (Free version of Claude literally would not let me send any more prompts in that chat log but would let me start a new one).

The automated testing request first tried to make me create a whole new racket project and run it from the terminal by extending my original project. I did not like this approach and as such, prompted it again requesting to keep the tests in the same file. This worked! And to my surprise, with zero errors and perfect tests! That covers testing.

## Code Breakdown:

I will go section by section of the code and give my opinions on the quality of code that Claude produced. Keep in mind, the code functions properly and correctly, so this will not focus on correctness, but rather the efficiency, readability, and overall comprehension.



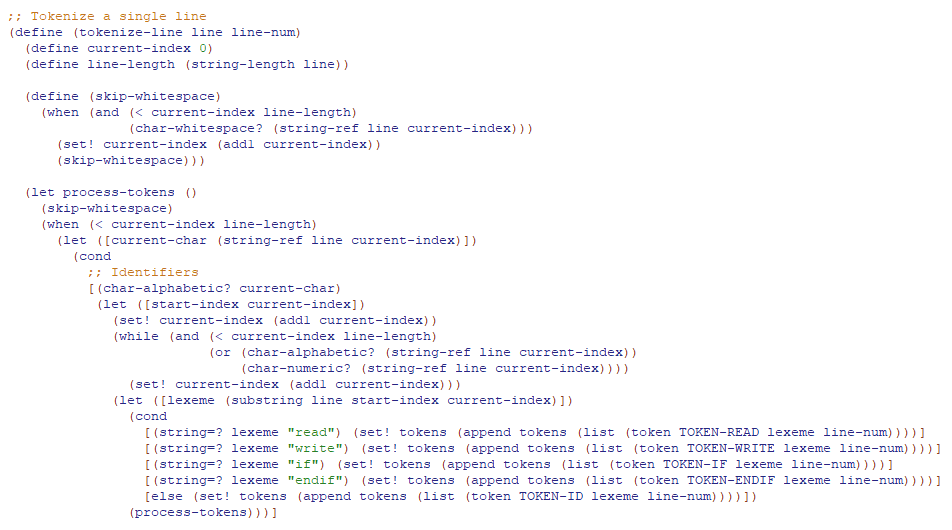
Definition of a new data structure, token. Seems straightforward and necessary, how I would code it. Claude did add the transparent tag which I wouldn’t have known to do. So I learned something new!

Token definitions. Nothing to note here.

Global definitions.

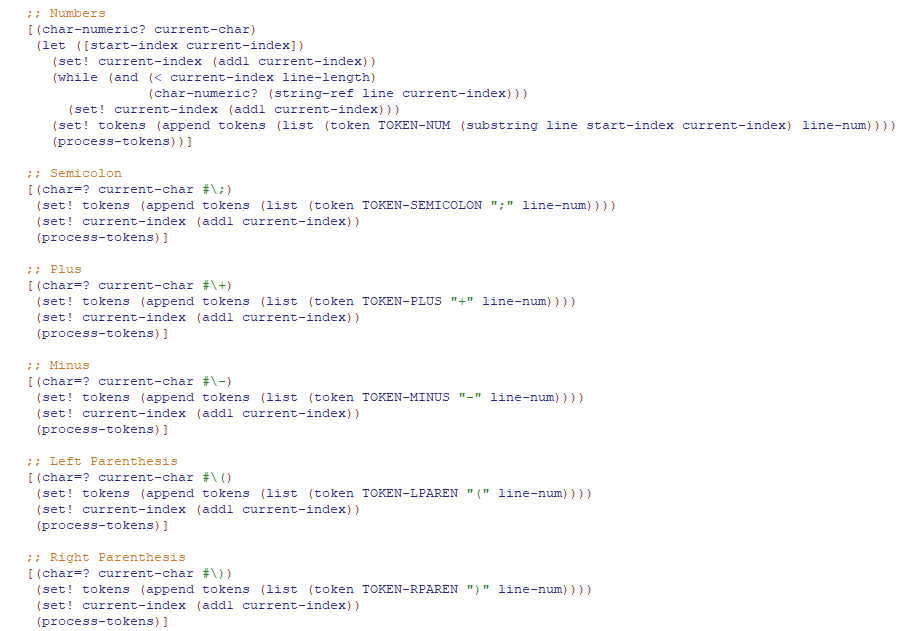
Here we have our first function! Tokenize takes an input string and turns it into a list of tokens. I love the name of this function and think it's exactly how I would code it.

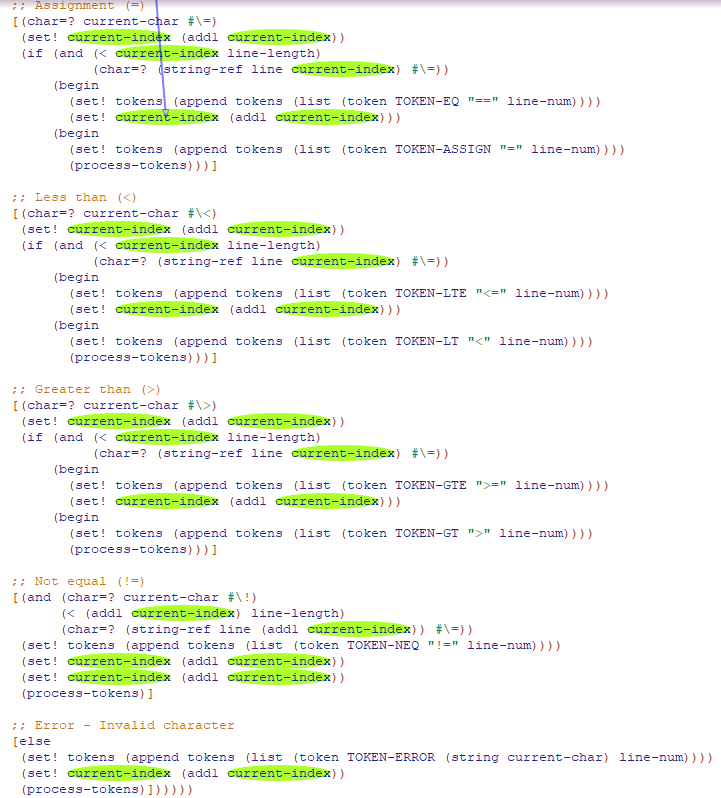
First part of tokenize-line



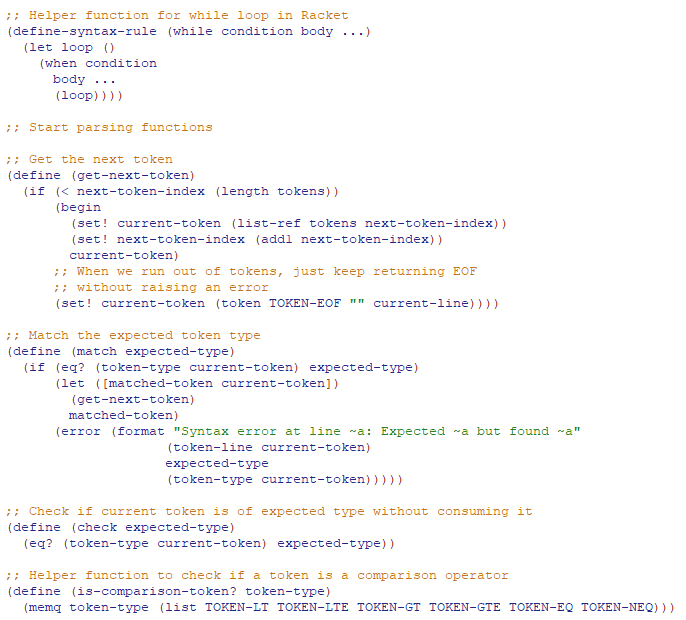
tokenize-line splitting up lines into their own function confused me at first. I don’t think I would’ve coded it this way, but looking at it, I do think it helps with both code readability and comprehension due to tokenize-line being a rather large function already. I know standard coding practices in OO programming call for functions to be as short as possible without sacrificing performability, I do not think functional programming is so stringent. But this size seems excessive even to me. If I were to code this by hand, I think having just this be tokenize-line and the next part be their own separate functions would provide better readability.

Second part of tokenize-line:





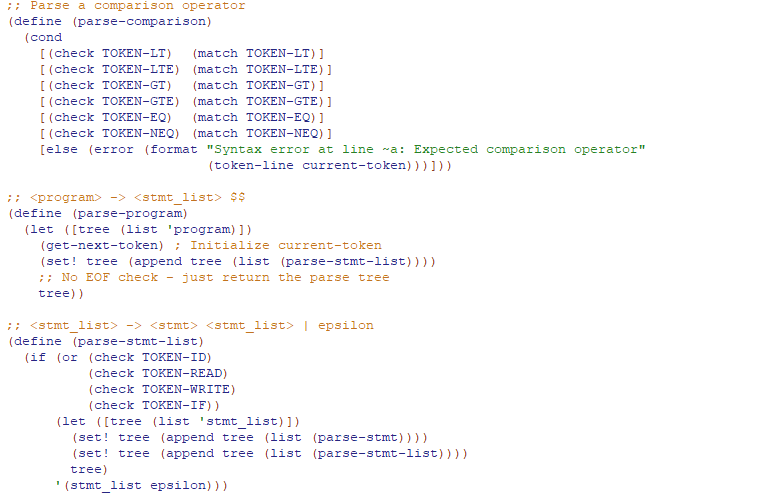
Here it outlines each of the lexemes, how to scan and then parse them using their token identifier established before. I think these are exactly what they should look like. I appreciate that Claude can easily code redundant, repetitive code that would be annoying for a developer to write out by hand.



This first helper function effectively creating a while loop is so interesting to me! I would not have taken this approach and, therefore, probably would’ve repeated this same code over and over again. Claude recognizing it would use this multiple times and making its own while loop function is really cool.

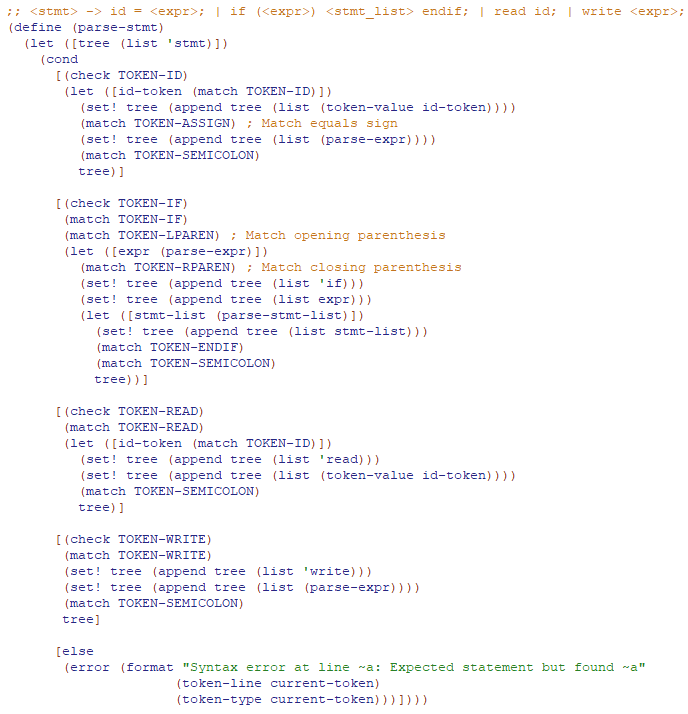
We’ve finally moved on from scanning to parsing! Get-next-token was originally breaking the code during the first iteration, which is why the eof statement was added. Other than that, its pretty straightforward and readable.

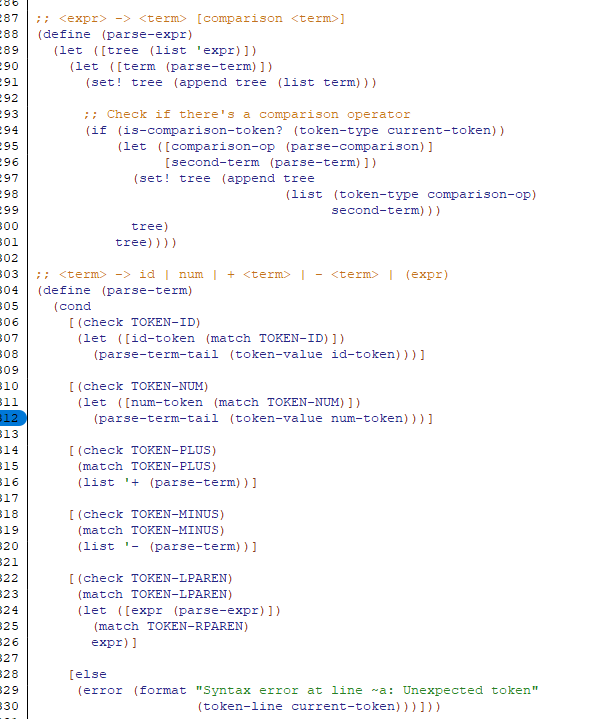
The match and check functions make up base parser logic, exactly how i would’ve coded them.

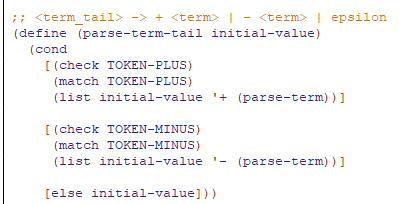


Parse-comparison was added in at a later step to handle the new compare nonterminal added in announcements. I placed it before the other parse functions for visibility.

The following are all the parse functions for each nonterminal. I don’t have much to say on them except I like the requirements telling us to keep each nonterminal its own function. Not sure if I would’ve done that and its absolutely necessary for readability. I also liked the comments Claude left above each of these functions to explicitly state the grammar that code is representing. Was very helpful to me as a developer. Lastly, it did get some parentheses messed up in this section. Not sure if it was due to the large amount of functions it was handling, but it was really only an isolated problem in this section of code.





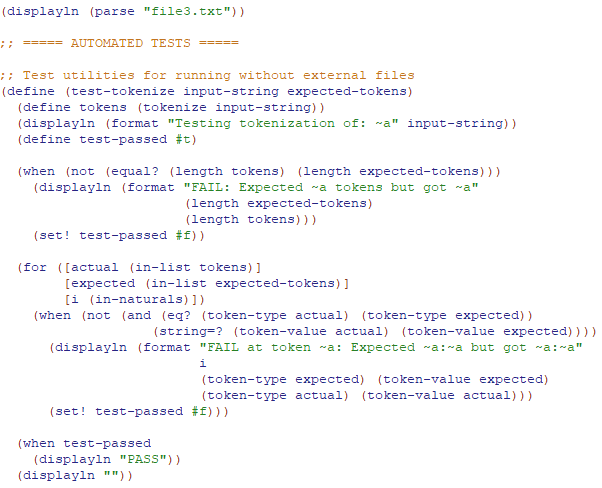


Here, we have the main function, the file to string function, and the running of the test file to actually start the code. I like how Claude left these at the bottom so they were easily identifiable for the developer. It was pretty easy to follow the logic of the code from these functions as a starting point. I also would have chosen to code this the same way, so no criticisms there.

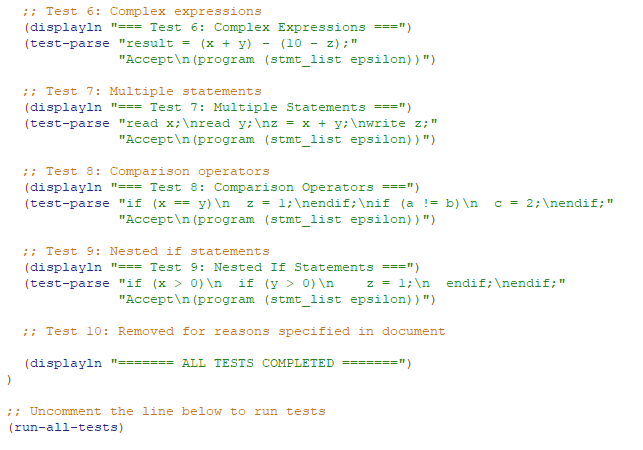
## Automated Testing:

I already showed manual testing with both a fake file made by Claude as well as the sample files provided to us by the professor. Here, I want to assess automated tests generated by Claude both for their performance and readability as well as to test my code.

Here is the code. It sits below the actual parser so it is out of the way and can be halted from running by commenting out its call at the end of the file. I did remove two tests that Claude generated, both because they had been covered by manually testing and specifically the 2nd test because it only wasn’t passing due to formatting of outputs but was correct otherwise.The formats for all the expected outputs also had to be changed but that was easy to do manually.

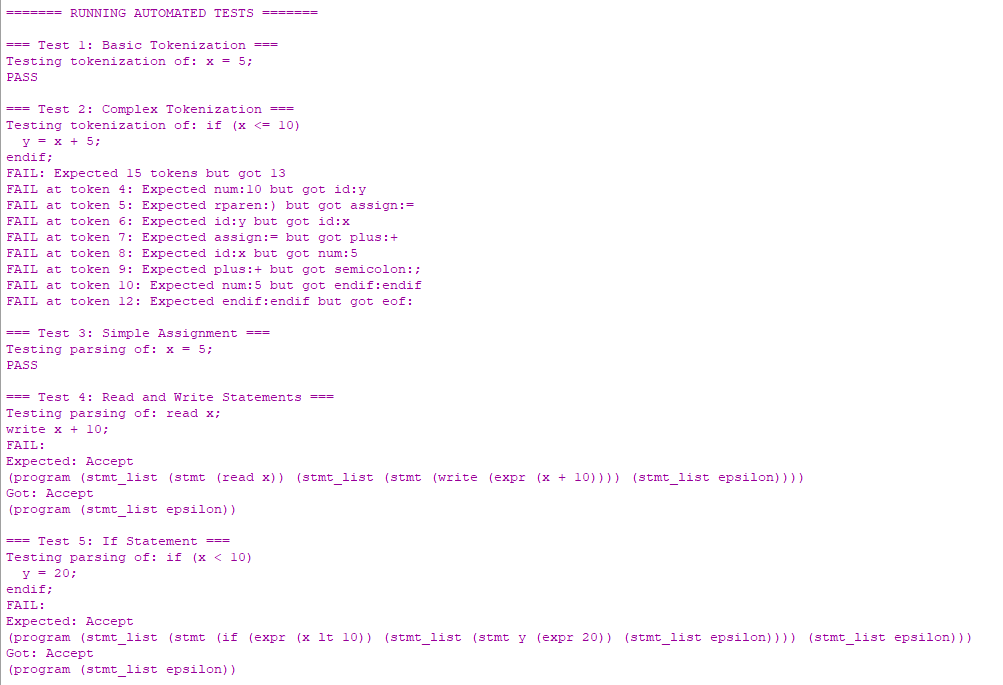


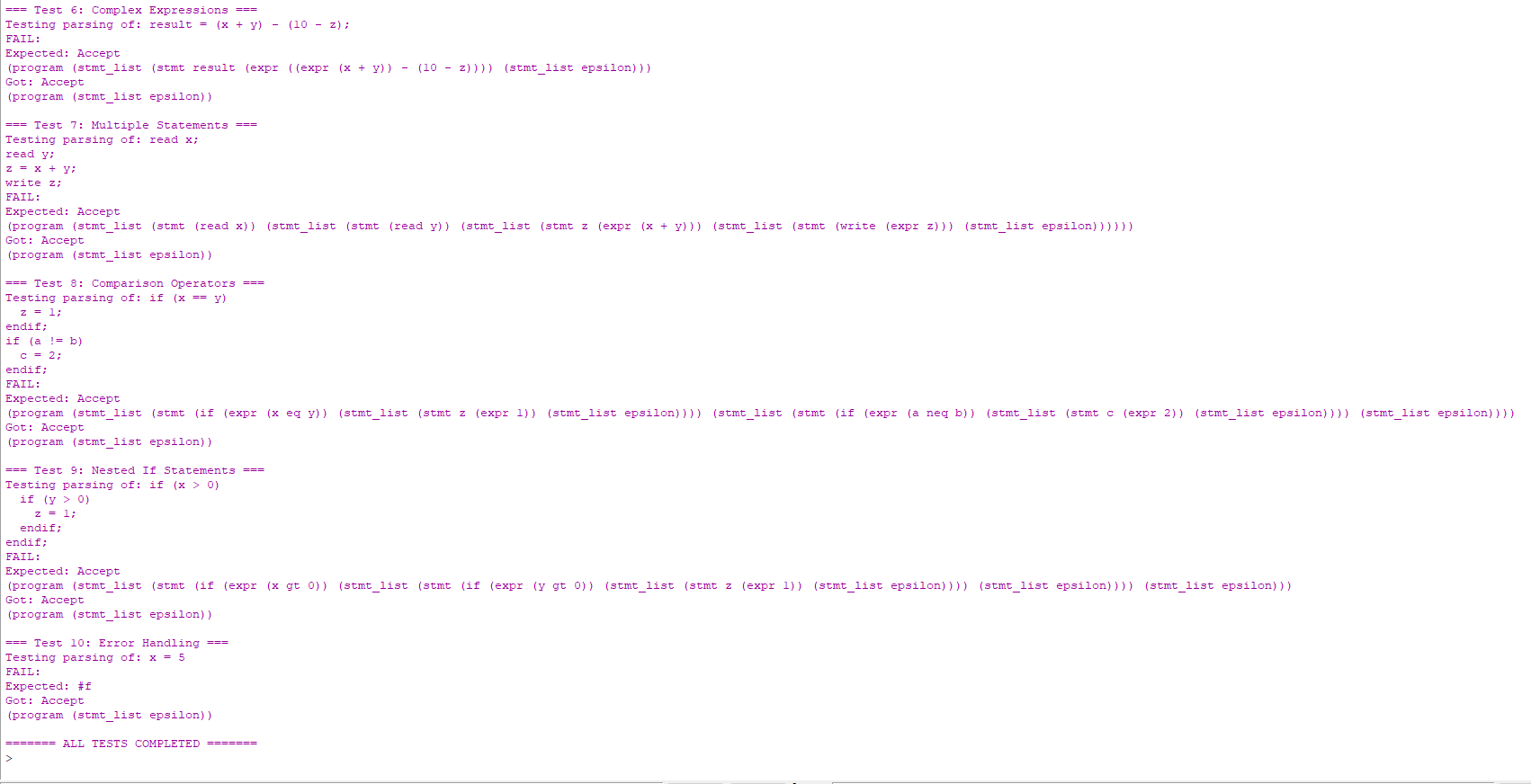




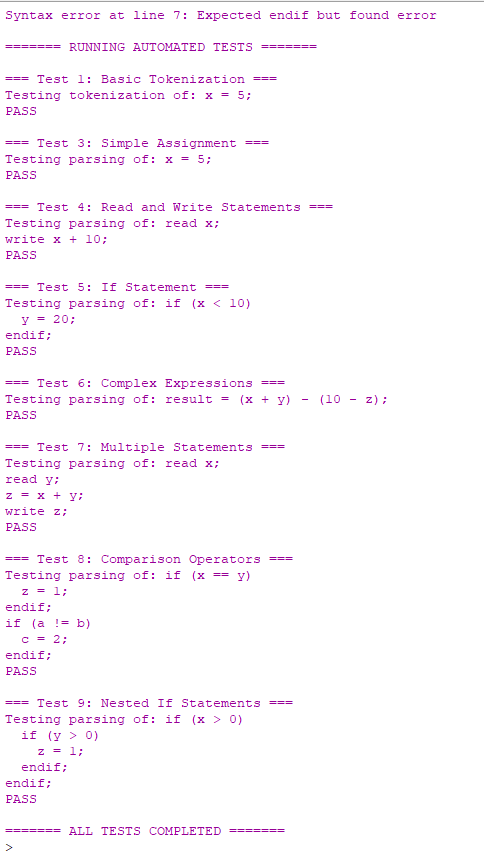
Very happy with how easy generating test code is, testing remains one of AI’s strong points as it doesn't really have to generate anything from scratch. Still, however, there were some issues that needed to be handled.

And here is the output of the tests before fixes





Output after fixes



All passing!

## Overview:

I think this project *really* showed Claude’s limitations to me again. The errors that occurred were minor and could be resolved. But the main issue is just how long this project was. Claude kept cutting me off and saying an individual chat was getting too long due to Claude unnecessarily pasting the whole code over and over again. I had to create 3 separate chats just to get the functioning code and an additional chat for testing.

Other than that, the code of Claude was solid. Minor errors here and there that could be fixed with a person familiar with coding practices.

These two projects have really proven to me that

1. AI is the future of coding
2. Coders will not be replaced by AI, but by people who can utilize it.
3. AI is far from being able to handle massive projects which is where the bulk of the industry is in. Sure scripts and short automation is nice, but the big companies hire coders to maintain decades old code with huge repos and thousands/millions of lines. Claude can barely handle the size of this parser with this short of grammar.